Synthesized Interaction on the X Window System

Krishna Bharat *Scott Hudson* *Piyawadee Sukaviriya*
GVU Center, College of Computing
Georgia Tech, Atlanta GA 30332-0280
`{kb, hudson, noi}@cc.gatech.edu`

Abstract
We have come to regard the human user, manipulating physical input devices as the sole driver of interaction in the graphical workspace. It is conceivable that for a variety of applications, such as help and tutorial systems, macro-by-example systems, session-playback systems and in collaborative work, we may require an alternative agent to perform tasks on the workspace, alongside the user. In this paper we describe a relatively non-intrusive and portable scheme for supporting such "synthesized interaction" on the X window system, and illustrate how toolkits may be instrumented to cooperate with such an agent at runtime, by providing information about the location of objects in their interfaces. In particular we describe the integration of synthesized interaction in the Artkit toolkit, which is structurally similar to most modern toolkits and should serve as relevant example.

Keywords: synthesized interaction, interaction techniques, animated help, event generation, X window system, X event, user-interface toolkits.

1. Introduction
Although we have come to regard the "simulated world" metaphor rather than the "conversational" metaphor as the dominant paradigm in user-interfaces [8], we continue to view the human user as the sole initiator of actions in the workspace. As in the real world, there is an opportunity in this simulated world to make life simpler by deploying general-purpose, programmable agents that can interact with interfaces and are capable of performing tasks, mimicing the user or demonstrating a facility. It is conceivable that in this closed world, such an agent, with easy access to knowledge about objects in the workspace, could actually out-perform the user in a variety of tasks.

1.1 Motivation
Consider the following scenario - an all too common occurrence in a world of changing software (see Fig. 1):

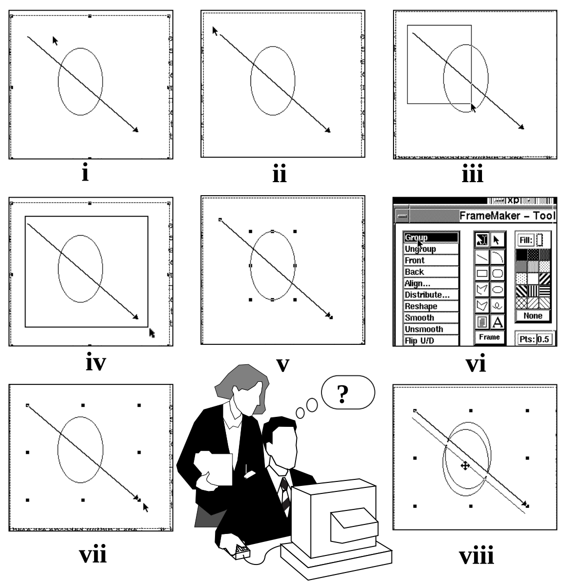


Figure 1. Demonstrational Help - Do we really need a human ?

Joe is learning to use a new editor and wants to know how to select multiple items. Fortunately, Joan is around and can show him how. She takes the mouse from him and walks him through the process. Not wishing to disturb his work, she creates a separate frame and draws a pair of objects - an arrow and an ellipse. She then says, "Watch, here I have two objects that I need to group." (box i) "I move to the top left corner." (box ii) "I press the left button and drag the mouse, and a little box appears." (box iii) "I drag the box until it encloses all the objects," (box iv) " and I release the button. Everything gets selected." (box v) "Then I go to the toolbox, and click on 'Group'" (box vi) "So, now they are all grouped together," (box vii) " and we can move them around.." (box viii) She then deletes the new frame and leaves his editor in its original state.

There is nothing that precludes a computer program from doing whatever Joan just did. She made use of no real world knowledge. All her actions could be translated into mouse moves, mouse button clicks and key presses. She needed to look at the interface often to find out where objects were and to make sure things were happening according to plan. She may have made use of objects in Joe's context to do the task more intelligently, but such state information and details of the interface layout are available with the application and the window-system, and should be within the reach of other programs. She punctuated her actions with verbal explanation, which could be recorded in audio files and played back.

In addition to the above *animated help* example, we could apply a synthetic interaction facility to a variety of tasks:

o Recording and playback of a session or a demo, tolerating minor changes in layout and at a different speed.

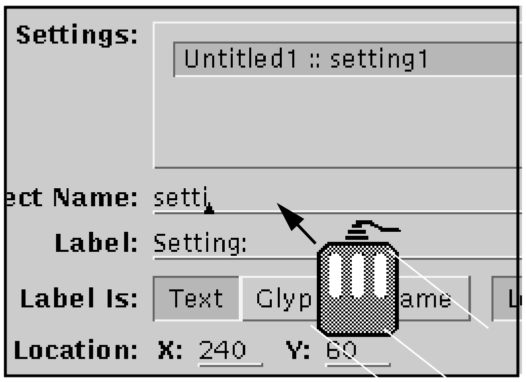
o Enabling handicapped users to operate on conventional interfaces with an agent driven by speech input.

o In Collaborative Work. Actions performed on one interface may be copied over and performed on another interface, at a later time.

o Task Automation. Given a simple scripting language users would be in a position to automate tasks, like starting up a tool and initializing its settings or repetitive actions like reading news or mail.

All we lack currently is the infrastructure to generate events, and a mechanism find out the location and status of objects in the interface and get acknowledgments from the application for actions that have been completed. In this paper we describe how these may be implemented.

Figure 2. Animation using keyboard and mouse events, moving icons and sound effects



Settings: Untitled1 :: setting1

ect Name: setti

Label: Setting:

Label Is: Text Glyp ame L

Location: X: 240 Y: 60

1.2 Prior Work

In an earlier paper [2], we motivated the need for such a facility and described the application programmer's interface to an agent called the *animation server*. The animation server was an independent process that would accept "animation" scripts from client applications and translate them into actions to be performed asynchronously. This involved synthesizing input events as if from the mouse and keyboard. It was capable of synchronizing with other media streams, and could query the client for the location of widgets during execution. A variety of applications were discussed.

An animated glyph (shown in Fig. 2) was used to represent the mouse (showing depressed buttons), with sound effects for button presses and key clicks.

We have subsequently dropped the heavily overloaded term "animation" in favor of "synthesized interaction," in appreciation of its non-demonstrative uses.

The focus on this paper is on event synthesis, security and interface related knowledge representation - issues which are orthogonal to and complement those addressed in [2], which dealt with the API to such a facility and its potential applications. In our earlier work we assumed the existence of an event generation mechanism, and did not explain how it would be implemented.

The organization of this paper is as follows. In Section 2 we evaluate a number of event generation techniques, and explain why we chose the technique that we have adopted. In Section 3 we describe a portable implementation in sufficient detail that it may be reproduced. Section 3.2 deals with implementing security at an event level. In Section 4 we present two schemes for making interface related knowledge available to other programs, and in Section 5 show how a typical toolkit may be extended to support these schemes.

2. Design

Implementing synthesized interaction on any window system is a fair challenge, and more so in the X window system. The lack of consistency among toolkits and the potentially unpredictable results of the asynchronous protocol contribute to the complexity. While our design is geared towards handling the hard problem, namely interaction in the X setting, it is equally applicable in a synchronous window system with a single toolkit. We address issues such as event dispatch, consistency between event streams, security and interface layout information which are relevant on any event driven window system.

2.1 The X Environment

In the X window system[16], the X server provides a virtual "display" abstraction and regulates access to a set of screens and input devices. Client applications communicate with the server using an asynchronous, networked protocol called the X Protocol, which is a standard for all releases of the X11 software.

The server displays windows on behalf of client applications, and dispatches X events to appropriate clients in response to input device events. X events are necessarily directed at specific windows. An application may have multiple “top-level” windows, which may in turn nest other windows corresponding to parts of the interface. A single input event can result in multiple X events, possibly to different windows - e.g. a single mouse move could cause an *exit* X event to be delivered to one window and an *enter* X event to be delivered to another.

The Xlib[11] library, which is an essential part of all windowing applications in the X environment, is responsible for all communication with the server. It queues the X events from the server as they come in, until they are ready to be processed by the client. In addition to being *notified* about changes in the status of input devices, the client can also *poll* the server using Xlib routines. Applications often deviate from their event driven execution to implement dragging, pulldown menus and rubber-banding by polling the state of the pointer.

2.2 Design Criteria

Events need to be generated at some level by an agent which we shall call the *interaction synthesizer*. Typically, this agent operates asynchronously with respect to the recipient and is best implemented as a separate process (or thread). In general, the synthesizer could operate on multiple applications on behalf of multiple clients.

In designing a synthesized interaction system we would like to keep the following criteria in mind:

(a) Portability: It should be possible to port the implementation to other platforms running X with relative ease.

(b) Application End Support: We would like open-ended support at the application end to find out the location of objects in the interface, get confirmation about executed actions, abort actions if necessary etc.

(c) Robustness: The transmitted event sequence should have the desired effect with a high probability. Getting feedback from the application about the processing of events will automatically contribute to robustness.

(d) Expressiveness: Any interaction sequence should be easily expressible in terms of the event primitives provided.

(e) Programming Overhead: The scheme should not impose a significant overhead on either the application programmer or the toolkit writer.

(f) Intrusiveness: The inclusion of synthesized interaction should not interfere with the normal operation of the application or window system in any way.

(g) Security: It should be possible to protect applications against events from unauthorized sources. Authorized users should be allowed to send events to select applications.

2.3 Level of Operation

There are many levels at which events may be generated:

(a) Device Event Level: Events could be introduced at the level of the device driver so that even the window server cannot tell they are fake. This could also be supported by an extension to the server. The IN^3 speech recognition software (Release 2) [4] does event synthesis at the device driver level. Our first implementation relied on the fact that the OpenWindows X server, *xnews* could accept program generated device events using the NeWS protocol. We rejected this approach for the following reasons:

1) It is either platform or server dependent and hence not portable.

2) It has been our experience that such event generation is too low-level to be robust. The synthesizer would transmit certain device level events to the window server expecting then to be mapped to the correct window system events, and sent to the appropriate windows. Unless the synthesizer replicates the server’s state there is no guarantee that this will actually happen. It is more robust to avoid this level of indirection and send higher level events directly to applications. Also, if we talk directly to the application we can get confirmation when events are processed.

3) It is hard to regulate access at a fine level of granularity, i.e. at the level of a individual windows. We would like to be able to allow authorized users to send events to select “interaction-friendly” windows and not to others.

(b) Application Event Level: Operation at the level of the application is a viable alternative. We could “magically” invoke callbacks within applications with the assistance of the toolkit. However this burdens the implementer of the toolkit considerably, and is subject to the following shortcomings:

1) This does not adequately simulate interaction since there is no visual feedback.

2) The implementation cannot be easily ported to other toolkits.

3) This requires a separate IPC mechanism for transmitting events to applications.

4) This could result in abnormal behavior since it circumvents the interface. For example, if we invoked the callback of a disabled widget there could be unpredictable consequences.

(c) Window System Event Level: Instead, we chose to operate at the level of window system events, or X events. There are three sets of events that directly result from user actions and need to be synthesized:

1) Events triggered by Pointer Motion
2) Events triggered by a Pointer Button
3) Events triggered by a Keyboard action

In the X window system these correspond to:

```
1) {MotionNotify, EnterNotify,
    LeaveNotify, FocusIn, FocusOut}
2) {ButtonPress, ButtonRelease}
3) {KeyPress, KeyRelease}
```

All other events in the system are the indirect consequence of input actions and get generated automatically as the interaction proceeds. The first set of events is easy to produce in X, since the pointer may be moved using the `XWarpPointer` routine in Xlib. The remaining need to be generated synthetically.

A simple way to send X events to the client is to use the `XSendEvent` facility in Xlib, which uses the `SendEvent` protocol request to get the server to forward an X event to an appropriate window. Unfortunately, this is not enough to synthesize interaction because:

i) The server sets a boolean field in such events to mark them as synthetic, causing most toolkits to ignore them. Some X servers allow the tagging to be turned off, but that would represent a serious breach of security.

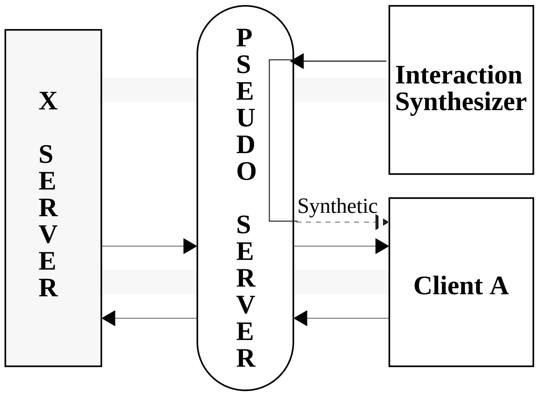
ii) Even if the events were processed by the toolkit, there is no way to prevent genuine communication from the server from conveying information about the state of the input devices that contradicts the impression created by the synthetic events.

iii) Applications often go into a polling mode. They query the server about the state of the pointer and the keyboard using the `QueryPointer` and `QueryKeymap` protocol requests. These need to be handled as well.

It seems inevitable that we must interpose at some point in the server-client communication. There are three ways that this interposition can be performed. First, the server could be modified to introduce new synthetic events at the source. Second, as illustrated in Fig. 3, a new "pseudo-server" process can be placed between the server and the client to introduce events at that point. Finally, as illustrated in Fig. 4, the client itself (in particular parts of the Xlib library used by all X client programs) can be modified to introduce events at the destination.

The first approach has been taken with the `Xtest` experimental extension included in X11R5. Server modification may pose a problem since it clearly requires the use of a specialized server. This can preclude running a vendor supplied server. Such a server may be highly tuned for particular graphics hardware, or in the case of some display hardware, may even be the only server available. For example, users (like many at our site) employing X terminals, rather than conventional workstations, cannot modify the server they use in any way. Secondly, this provides no support to monitor and control the processing of events at the application end.

Figure 3. Adding Synthetic X Event Packets to the Protocol Stream

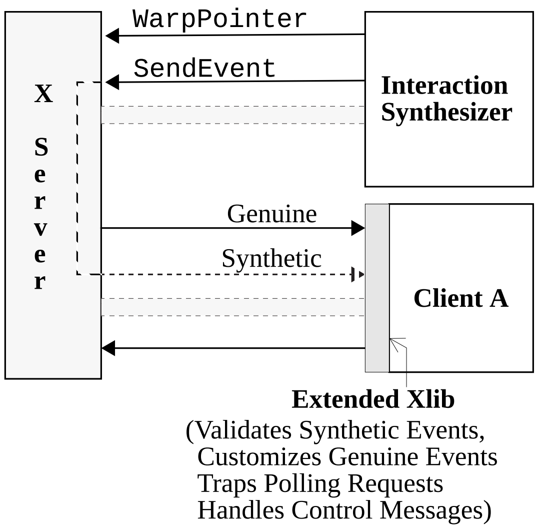


Interposing with a pseudo-server between the client and server offers what, at first, looks like a cleaner solution, requiring no modification to existing components. Client programs simply connect to the pseudo-server (e.g. by changing their DISPLAY environment variable) rather than the normal server. Systems like XTV[1] use the pseudo-server technique to implement shared windows. The main problem with this approach is performance. Since all communication is being intercepted, output requests which are critical to performance get needlessly delayed. While the scheme is non-intrusive, as in the previous case, it provides no support at the application end.

The final alternative, and the one we eventually settled on, is to interpose on the client side (see Fig. 4). To avoid replicating the functionality of the server, this approach directly involves the server using the `WarpPointer` and `SendEvent` mechanisms, that are part of the standard protocol suite. All movement events are generated by

`WarpPointer` (which forces the server to generate all needed window *enter*, *exit* and *focus* events) and handled like other movement events. All other events are sent via the standard `SendEvent` mechanism. As described in the next section, the client, after validation of synthetic events (see Section 3.2), resets the `send_event` flag, updates a small amount of state-tracking bookkeeping (so events report the apparent rather than actual state of devices), and allows the events to be processed as usual.

Figure 4. Using the `SendEvent` Protocol and an Extended Xlib to Introduce Synthetic Events.



While this approach is more intrusive than we would like -- it requires a (very small) modification to the Xlib code used by the client -- we have found that many of the more interesting uses of synthetic interaction are better performed with cooperation of the client's toolkit anyway. When this is the case, intrusion is limited to the client portion of the overall system rather than being spread between the client and server. In addition, on systems that support dynamic linking, it is possible to interpose very simply without recompiling, relinking, or modifying the client in any way. For example, on Sun systems we merely need to retarget LD_LIBRARY_PATH environment variable to point to the extended Xlib.

The scheme is robust. We can target events at specific windows and get confirmation when they are processed. If necessary we can abort unprocessed events or filter out events from the human user, which could interfere with the interaction. It is highly portable. We ride on the X Protocol and require no additional locale dependent communication channels. Also, we are in a position to implement security at the level of individual windows. *The best reason for choosing this approach is that it provides virtually unlimited support at the application end for enhancements.*

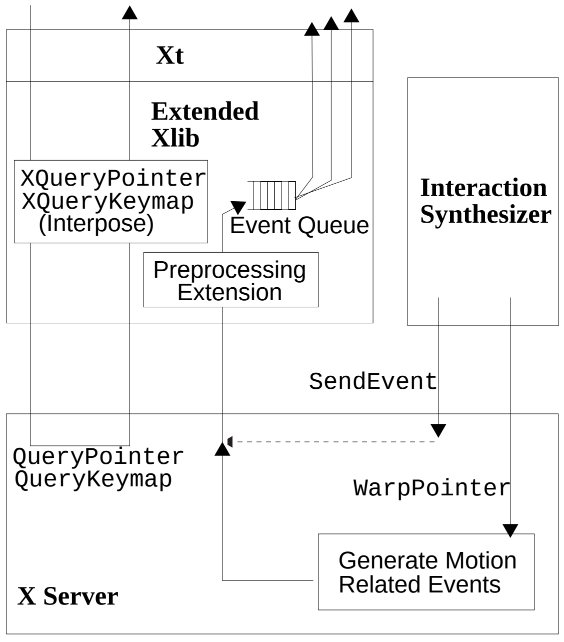
3. Implementation

3.1 Extending Xlib

Events may be intercepted either when they are enqueued for the first time, or when they are taken off the queue. The former approach is more practical because Xlib provides an extension mechanism for preprocessing events as they are enqueued. A more subtle reason for taking this approach is that events get enqueued in the sequence they are delivered (consequently with increasing timestamps), while the client may dequeue events out of order. Since synthetic events may cause the modification of events that follow, we need to ensure that they are processed in sequence. The synthesizer may also send control messages to the Xlib support code. These are encoded within a special X event called `ClientMessage`. Note that we could use a control message to operate on an event in the "past" if it has yet to be dequeued. This could be used for instance to abort actions by removing all pending events.

The figure below gives a detailed picture of the extension:

Figure 5. Extending Xlib



The Xlib source is publicly available from FTP sites such as *public.x.org* (in `/pub/R5untarred/mit/lib/X`). Fig. 6 shows the event preprocessing code in the extension. Synthetic events are first subject to an authorization check (line 2). This is discussed in Section 3.2.

Authorized synthetic events of type `ClientMessage` (lines 5-7) carry control information. They have a 32 bit type field and a payload of 20 bytes for data. `ClientMessages` can be used to set up a general-purpose RPC

Figure 6. Extension for Preprocessing Events as they are Enqueued

```
1   if (event.send_event)     /* event.send_event is set in synthetic events */
2      is_authorized := authorization_check(event);
3      /* Check if the event is from an authorized source */
4      if (is_authorized)
5         if (event.type == ClientMessage)
6            ...          /* Treat as a control message. Take appropriate action */
7            return DONT_ENQUEUE;/* Do not enqueue this event */
8         else
9            event.send_event := false; /* Turn off the tag */
10           /* The only events that get here will be {ButtonPress, ButtonRelease,
11                                                  KeyPress, KeyRelease} */
12           event.state := Apparent_Input_Status;
13           Update_Input_Status(Apparent_Input_Status, event);
14           /* Maintain the apparent state of buttons and modifiers internally */
15           event.time := Logical_Time_Value(event, Local_Clock);
16           /* Compute a logical time value for the event and advance the local clock */
17        endif
18     endif
19  else /* Not a synthetic event.. */
20     if (event.type in {MotionNotify, EnterNotify, LeaveNotify})
21           /* Motion related events that carry state information */
22           event.state := event.state | Apparent_Input_Status;
23     endif;
24     Local_Clock := Advance_Local_Clock(Local_Clock, event);
25     /* If the event has a time field it advances the local clock */
26  endif;
27  return ENQUEUE; /* Enqueue event as before */
```

facility, and implement a variety of functions, some of which may require support from other parts of the library. This may be used for instance, to set a mode wherein certain types of events are filtered out or modified, e.g. button and keyboard events from the user which could interfere with the interaction. In the same manner we could request notification when a certain batch of events has been processed, for synchronization. We use this mechanism to enable the interaction synthesizer to discover the location of objects in the interface, as described in Section 5, using "upcalls" to the toolkit and application layers.

If the synthetic event is not a `ClientMessage` (lines 9-16), it must be a button or key event, destined for the toolkit. The `send_event` tag is turned off. Most event fields such as the destination window, location within the window and the button or key involved are set by the interaction synthesizer. Two fields still need to be set.

1) `event.state` (line 12), which holds the state of the buttons and modifiers before the event occurred. For this purpose a state vector called `Apparent_Input_Status` is maintained (line 13). It represents the state of the buttons and modifiers, as shown by the synthetic event stream and gets updated by each synthetic event.

2) `event.time` (line 15), which holds a timestamp for the event measured in milliseconds. This is assigned by the X server, except in synthetic events where it needs to be computed by the interaction runtime. In most cases this value does not matter. Consequently, the synthesizer initializes the field to a special value, `DONT_CARE`, and the preprocessor reassigns it to a value that is marginally greater than the last genuine timestamp seen. For this purpose a variable called `Local_Clock` is maintained. In special cases the relative time between events is critical. For instance, toolkits look at the timestamps of button events to distinguish between a click and a drag, and two clicks and a double-click. In these cases the synthesizer assigns values to the `time` fields to represent constraints between successive timestamps. This causes the preprocessor to assign timestamps that respect the constraints,

ignoring the `Local_Clock` value. For example the events:

```
ButtonPress 0
ButtonRelease 100
ButtonPress 300
ButtonRelease 400
```

will be converted to

ButtonPress t_0
ButtonRelease t_0+100
ButtonPress t_0+300
ButtonRelease t_0+400

where t_0 is the value of `Local_Clock` when the first event is received. The above sequence could represent a double-click. The bounds are toolkit dependent and cannot be hardcoded in the preprocessing code.

In the case of non-synthetic events (lines 20-25), book-keeping is done to maintain `Local_Clock`. If the event is `MotionNotify`, `EnterNotify` or `LeaveNotify` it has a `state` field representing the state of buttons and modifiers. A bitwise OR operation (line 22) is used to ensure that the field reflects buttons and modifiers depressed during the synthesized interaction, as shown by the state vector `Apparent_Input_Status`. Note that when a synthesized interaction sequence is not in progress the `Apparent_Input_Status` vector will be empty and this operation will have no effect.

The routines `XQueryPointer` and `XQueryKeymap` return vectors representing the status of buttons and keys, and need to modified so that they return a value consistent with the `Apparent_Input_Status` vector.

3.2 Security

The introduction of a synthetic event generation capability, while offering important new functionality, also has serious security implications. The unencrypted X protocol is only as secure as the underlying transport mechanism and operating system, and there are a number of attacks possible against a typical X window system set-up. Our aim is to ensure that the validation of synthetic events by the extended Xlib does not open any new security loop-holes. Further, we would like this guarantee to hold true even in the future, as the rest of the system becomes more secure.

Attacks may be mounted at various levels:

Level 1: The infiltrator is able to introduce seemingly genuine events.
Level 2: The infiltrator can eavesdrop and delay, stop, or reorder the delivery of events.
Level 3: The infiltrator is only able to eavesdrop.

Note that infiltrators who are able only to modify events in the event stream, can themselves send synthetic events using `XSendEvent`, then intercept them and clear the `send_event` bit to create a seemingly genuine event of any sort they choose. As a result, the ability to modify events alone is equivalent to a Level 1 attack. Without additional support from the underlying transport system, there is no way for a client to distinguish a Level 1 attack from legitimate messages.

A Level 2 infiltrator can cause harm by capturing messages and later replaying select portions, or simply by removing select input events, (for example changing the key-press sequence for “`cd obj; rm *.o`” into “`cd ; rm *`”). It might seem at first that we need to guard only against Level 3 infiltrators since the other variety could operate just as easily on the genuine event stream. However, the threat from a Level 2 attack is considerably higher in the case of synthetic events because genuine events are under considerable human supervision.

We will outline a scheme that guards against Level 2 attacks -- an infiltrator who may be able to eavesdrop, stop, reorder, replicate, modify (by stopping and retransmitting), or introduce events in the synthetic event stream.

Proper security requires that the party sending a synthetic input event can always be identified by a signature so that events generated by unauthorized parties can be rejected. It should not be possible to forge the signature, nor should it be possible to affix the signature on any other event or transmit events out of order. The information content of the signature should therefore be derived from the text of the event and a message sequence number in a non-reversible fashion. To prevent forgery this information can be encrypted using an encryption key known only to the receiver and authorized sender. This key could be known a priori to both parties (for example, in the most common case when the sender is an agent process executing under the same user id on behalf of the receiver), or may be exchanged at run-time. A secure key exchange between parties not sharing a prearranged key or other secret can be performed by well known public-key cryptography methods [13, 5].

In the case of button and key events there is a 32 bit sub-window field which is always set to zero in synthetic events. Consequently, this field can be made to hold the signature in transit. `ClientMessage`s carry 20 bytes of data, 4 of which can be used for the signature.

The encrypted data is hashed to a 32 bit signature to fit in the available payload as follows,

signature := hash32(crypt(key, event + seq))

where crypt() is an encryption function, seq is the sequence number of the event, and hash32() hashes a longer string into a 32 bit value.

A slightly more efficient alternate approach which encrypts only 32 bits of plaintext for each message would be to use:

signature := crypt(key, hash32(event + seq))

Note that this scheme requires relatively strong cryptographic properties for the encryption function, since the plaintext and (at least a function of) the cyphertext are both always available. Similarly, the hash function used must be relatively well distributed and, in the second scheme, have the property that all bits of its input affect its output (for a discussion of more sophisticated hash functions for digital signatures see for example [14, 15]). On the other hand, most synthetic interaction sequences would be relatively short with respect to the length of the key that could be exchanged, and would be expected to last only a few minutes. In addition, the system could be set up to “time-out” after a few minutes of inactivity. As a result the time available for an adversary to apply the computations needed for a cryptanalytic attack is in practice quite small. This will result in relatively strong security (at least significantly better than other aspects of a typical X window and Unix system) even when relatively simple cryptography is employed.

4. Specifying Interaction

The synthesizer is structured as two layers. An object oriented toolkit layer, which is responsible for event generation, and an interpreter for a scripting language, which drives the toolkit. The toolkit can be used to create objects that manage the pointer and the keyboard on a certain display. The objects may be subclassed to change their appearance or behavior. This is how we integrate animation techniques such as slow-in and slow-out and motion along a curve. The interaction synthesizer is implemented as both a library and a stand-alone program.

When asynchronous execution is desired clients can start up the synthesizer as a separate process and drive it with interaction scripts. Scripts are communicated using a general purpose RPC facility called Intercom, also implemented using the `SendEvent` protocol.

4.1 The Scripting Language

The scripting language resembles the one described in [2]. An interaction technique that operates on a scroller would look as follows:

```
Goto CENTER @scrolr1.thumb in 800 msecs
Press Left
Goto NORTH @scrolr1 + 0,50 in 2 secs
Release Left
```

The notion of time has been incorporated. Movement can be expressed in terms of a duration. The synthesizer uses an adaptive technique to compute the intermediate steps along the path. It computes the time elapsed at intermediate points and dynamically adjusts its velocity to ensure that the deadline is met [7]. The motion is smooth on a fast machine and gracefully degrades on slower machines. This makes it easy to synchronize the interaction with audio playback.

Actions are expressed using destination names. Button and keyboard actions, and pointer moves are specified in terms of application specific destinations, rather than absolute coordinates or X window ids which are likely to change over time or over invocations. In the above example, “`NORTH @scrolr1 + 0,50`” refers to a point that is offset by 0,50 from the center of the northern edge of an object called `scrolr1`.

The interpreter uses such expressions to compute the screen locations where the events should have occurred. In the case of a button or keyboard event it also provides the destination X window, known as the event focus. In some cases the destination is implicit as in the Press and Release statements above, and the interpreter uses the previous destination. Providing useful defaults and such implicit behavior hides the details of the event delivery from the script-writer. Unfortunately this does not work in all cases.

In the above example they would need to set the focus for the Release event explicitly using

```
Focus @scrolr1.thumb; Release Left
```
or
```
Release Left to @scrolr1.thumb
```

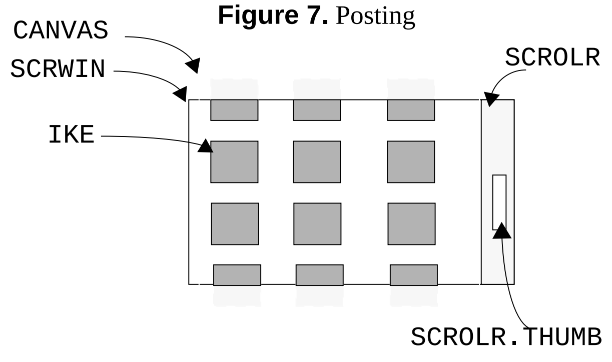
since the Release event should logically be delivered to the thumb of the scroller. In many UI toolkits this would not matter, either because `scrolr1` and `scrolr1.thumb` would share the same X window, or because the thumb would have “grabbed” the pointer upon the Press event and would get the Release event in any case.

4.2 Naming Locations

To intelligently operate on the client's interface, the synthesizer needs information about the location of objects. Note that these objects may or may not correspond to an entire X window. In general they could represent a subdomain (we assume rectangular) within a window. In Motif [12] and other Xt based toolkits there is a close correspondence between X windows and widgets. For retargettabilty and control over event delivery toolkits like InterViews [9], Artkit [6] and Trestle [10] implement their own nesting scheme, and all widgets reside within a single top-level X window. Consequently locations are expressible as subdomains of the same window. In either case there is a nesting of rectangular regions, and the synthesizer needs to discover the smallest enclosing X window corresponding to a given destination and compute the latter's location in the window's coordinate system. There are two ways in which the client can make such information available:

1. *Querying,* where the synthesizer asks the client by interprocess communication. Since the synthesizer already communicates with the client using the `SendEvent` protocol, the query can be placed in a `ClientMessage` and will be handled by the extension. The reply is sent in a similar manner.

Figure 7. Posting



`IKE`:= (`CANVAS`, NW + (Delta_x, Delta_y, Wid, Ht))
`CANVAS` := (`SCRWIN`, NW + (dXc, Scroll_y, Wc, Hc))
`SCRWIN` := (`0x1239a7`, NW + (dXs, dYs, Ws, Hs))

2. *Posting,* where it is posted in a public place. A good place to place labeled, shared information is in a known X window. X allows arbitrary "properties" to be attached to windows. The synthesizer could lookup the location of objects by name on a special "posting" window of the client it is interacting with.

We support both schemes and let the implementer of the client or the toolkit choose a scheme that suits them. If querying is adopted it is most convenient for the client to do all the computation and return values in screen coordinates. In the case of posting, rather than post everything in screen coordinates and suffer the overhead of re-posting when a top-level object is repositioned, we are able to post relative locations. The location of an object is expressed as a function of another object, typically its parent in the toolkit's nesting scheme.

To illustrate this consider an application with multiple icons in a scrolling window, shown in Fig. 7. The location of an icon `IKE` is expressed as an offset from the top left corner of the canvas pane called `CANVAS`. This offset is constant as the window scrolls and does not need to be reposted. `CANVAS` is posted at a displacement from `SCRWIN,` a scrollable window interactor, which is posted relative to the enclosing X window, 0x1239a7. As the interactor scrolls the client need only re-post the location of `SCRWIN`. While this requires more lookups on the part of the synthesizer, the overhead is negligible when compared with human cognition time. In practice it is necessary to insert pauses between steps in the interaction script to slow down the action and make it seem natural!

With posting, we can make use of various encoding schemes to specify the location of an object in terms of others. Some toolkits provide constraint or gravity based relationships.

Note that there may be multiple, possibly overlapping regions of interest within the same interactor. The regions `SCROLR` and `SCROLR.THUMB` are parts of the scrollable window interactor and also need to be posted. The synthesizer treats every name as a string and the naming convention is left to the discretion of the author of the application. The script-writer need of course be aware of the convention and the names used.

Posting is inherently more complex to implement than querying, since we need to trap changes to region-attributes. However the posting mechanism makes it possible for one application to make a large body of dynamically changing information (which need not necessarily be geometric or even interface related), available to potentially, multiple applications that would like to use the knowledge for reasoning. This is a fascinating prospect.

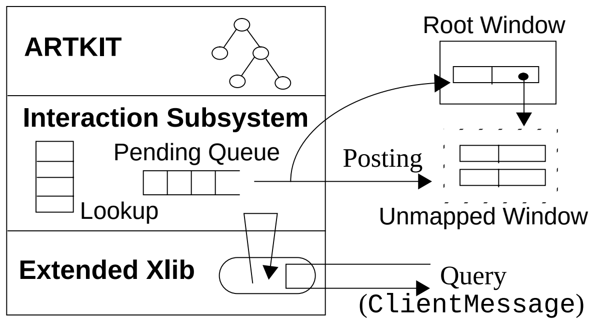
5. Toolkit Support

We describe how to extend a toolkit to support posting and querying, using the specific example of the Artkit toolkit[6]. Artkit is an object oriented UI toolkit on the lines of InterViews[9] and other modern toolkits. Each window in the application is implemented by a tree of interactive objects, called *interactors*. Every interactor draws itself relative to its parent in the tree. A single top-level X window contains all the interactors in a given tree. All classes of interactors are implemented as derived classes of a single base class called `min_interactor`. This is a blessing, since most of the extensions we need to make may be done at this level and will be inherited by derived classes automatically.

Posting and querying is done in terms of *regions*. A region is either a full interactor (such as a scroller) or a part of it (such as the thumb of the scroller). Fig. 8 shows the structure of the Artkit extension. The interaction subsystem maintains a lookup table containing references to all known regions. The application programmer makes a region known by associating a name with the interactor that contains it. This causes the interactor to create names for all its parts using a naming convention (currently we append a suffix to the name such as '.thumb' or '.top'), and register them in the lookup table. Since posting is done relative to the parent, an interactor will ensure that its parent is registered before it register its own regions. This may require the creation of a unique name if one has not already been provided.

To support querying, the interaction subsystem registers a callback with the extended Xlib, and gets notified when a query is received (within a `ClientMessage`). In response to the query the subsystem looks up the appropriate interactor and asks it to compute the dimensions of the region relative to its smallest enclosing X window. In the case of Artkit there is just one. The computed coordinates are returned in a `ClientMessage`.

Figure 8. The Artkit Extension



To support posting, the interaction subsystem maintains a pending queue, containing regions that have changed and need to be posted. When a region discovers that it has changed, it marks itself as dirty and adds itself to the pending queue. The dirty bit is cleared when the queue is flushed next. Since no region will actually appear to have changed until it redraws itself, we flush the pending queue only when redraws happen. This will allow multiple changes to a region to be compacted into a single posting. Detecting when a region has changed is vital to the success of the posting scheme. Fortunately, in Artkit, the `min_-interactor` class regulates access to the position, location and parent fields, by corresponding methods. This makes it easy to trap changes. In the case of a region that corresponds to a part of an interactor, we need to modify the derived class. A simple strategy to handle parts is to create a single `check_and_post()` method for the interactor to check if any of the parts have changed, and place triggers in various other methods that cause it to be invoked.

Since there may be multiple instances of the same application running, the posting is done on a special unmapped window created by interaction subsystem for that purpose. The application programmer is expected to supply each instance of the application with a unique name. This is posted on the root window along with the window id of the unmapped window. The interaction synthesizer first does a lookup on the root window to find out where data from a certain instance gets posted, and then performs further lookups on that window.

6. Conclusions

In the induction of any new technology, there are two forces at work. On the one hand, we have potential applications of the technology which motivate its development. On the other, we need to consider the overhead it brings with it and the structural changes that are called for. The questions - Will it jeopardize the robustness/security of existing systems? Is it a portable solution? Does it have scope for future enhancements? - need to be answered.

In [2] (and to a smaller extent here) we have motivated the need for a synthesized interaction facility with a set of potential applications. We still needed to address the issues of portability, security, robustness and extensibility, which we have done in this paper. We do not believe that this is the last word on event synthesis. It is likely that a standard server extension in the future will provide all the event generation support that we need. However there is no doubt that we need intelligence at the client end to provide feedback and close the agent-application communication loop. Hence we believe that much of this work will have relevance in the future, and in other windowing systems.

7. References

[1] Abdel-Wahab, H.M. and Feit, M.A. XTV: A framework for sharing X Window clients in remote synchronous collaboration, in *Proc. IEEE Conf. on Communications Software: Communications for Distributed Applications & Systems*, 1991, pp. 159-167.

[2] Bharat, K. and Sukaviriya, P. Animating User Inter-

faces Using Animation Servers, in *Proc. UIST* ‘93, pp. 69-79.

[3] Chang, B. and Ungar, D. Animation: From Cartoons to the User Interface, in *Proc. UIST* ‘93, pp. 45-55.

[4] Command Corp. Inc., *Private Communication*.

[5] Diffie, W. The First Ten Years of Public-Key Cryptography, in *Proc. of IEEE* v76, 1988, pp. 560-577.

[6] Henry, T.R. and Hudson, S.H. Integrating Gesturing and Snapping into a User Interface Toolkit, in *Proc. UIST* ‘90, pp. 112-121.

[7] Hudson, S.E. and Stasko, J.T. Animation Support in a User Interface Toolkit: Flexible, Robust, and Reusable Abstractions, in *Proc. UIST* ‘93, pp. 57-67.

[8] Hutchins, E.L., Hollan, J.D. and Norman, D.A. Direct Manipulation Interfaces, in *User Centered System Design*, Ed. Norman, D.A. and Draper S.W. Lawrence Erlbaum Associates, Ch. 5, pp. 87-124.

[9] Linton, M., Vlissades, J.M. and Calder, P.R. Composing User Interfaces with InterViews, in *IEEE Computer*, Feb 1990 22(2) pp. 57-67.

[10] Manasse, M.S. and Nelson, G. Trestle Window System Tutorial, in Nelson, G. (Ed.) *Systems Programming with Modula-3*, Prentice Hall.

[11] Nye, A. (Ed.). *Xlib Programming Manual*, O’Reilly & Associates Inc., 1992, Vol. 1, 3rd Edition.

[12] Open Software Foundation, *OSF/Motif Programmer’s Reference*, Release 1.2, Prentice Hall, 1993.

[13] Rivest, R.L., Shamir A., and Adleman, L. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems, *CACM*, Feb 1978 21(2), pp. 120-126.

[14] Rivest, R.L. The MD4 Message-Digest Algorithm, in *Crypto ‘90,* Springer-Verlag, 1991, pp. 303-311.

[15] Rivest, R.L. RFC 1321: The MD5 Message-Digest Algorithm, *Internet Activities Board*, April 1992.

[16] Scheifler, R. and Gettys, J. The X Window System, in *ACM Trans. on Graphics*, April 1986, 5(2) pp. 79-109.